TRANSLATORS

ATHANASIOS ROUDIS 5098
STYLIANOS SIMANTIRAKIS 5127

CONTENTS

1) Method of execution

The compiler file is : cpy.py .To run the compiler you need to open a terminal in the folder where you saved cpy.py and run the following command:

\the_path_of_the_file>/python cpy.py nameOfTheFileToTest.cpy

The code used to make this command possible is shown in Figure 1.

```
#Gia to diabasma apo ta arguments
parser = argparse.ArgumentParser(description='Compile a file.')
parser.add_argument('input_file', help='Path to the input file')

args = parser.parse_args()

fileToCompile = args.input_file

try:
    f = open(fileToCompile)
except Exception as e:
    print(e)
```

Picture 1

THE intermediate code is produced in the file intermediate-for-(nameOfTheFileToTest.cpy).int and the symbol table is generated in the file symbol-table-for-(nameOfTheFileToTest.cpy).sym .The final code is generated in the file assembly-for-(nameOfTheFileToTest.cpy).asm.

2) Files to check correct operation

The files for the health check are:

i) test.cpy , the given file to test.

ii) onlyMainTest.cpy , contains a program with only the main function.

iii) moreThanOneDeclarationsTest.cpy, contains a program with more than one declarations.

iv) limitsTest.cpy , contains a program with limit values.

v)smallTest.cpy, contains a program from the slides.

vi)ifWhileTest.cpy , contains a program from the slides .

vii) finalCodeExampleTest.cpy , contains a program from the slides .

3) Verbal analyzer

To create the verbal analyzer, the automaton of Figures 2,3,4 & 5 was used. More specifically, the parser reads the characters from the input file one by one and decides which state of the automaton it is in.

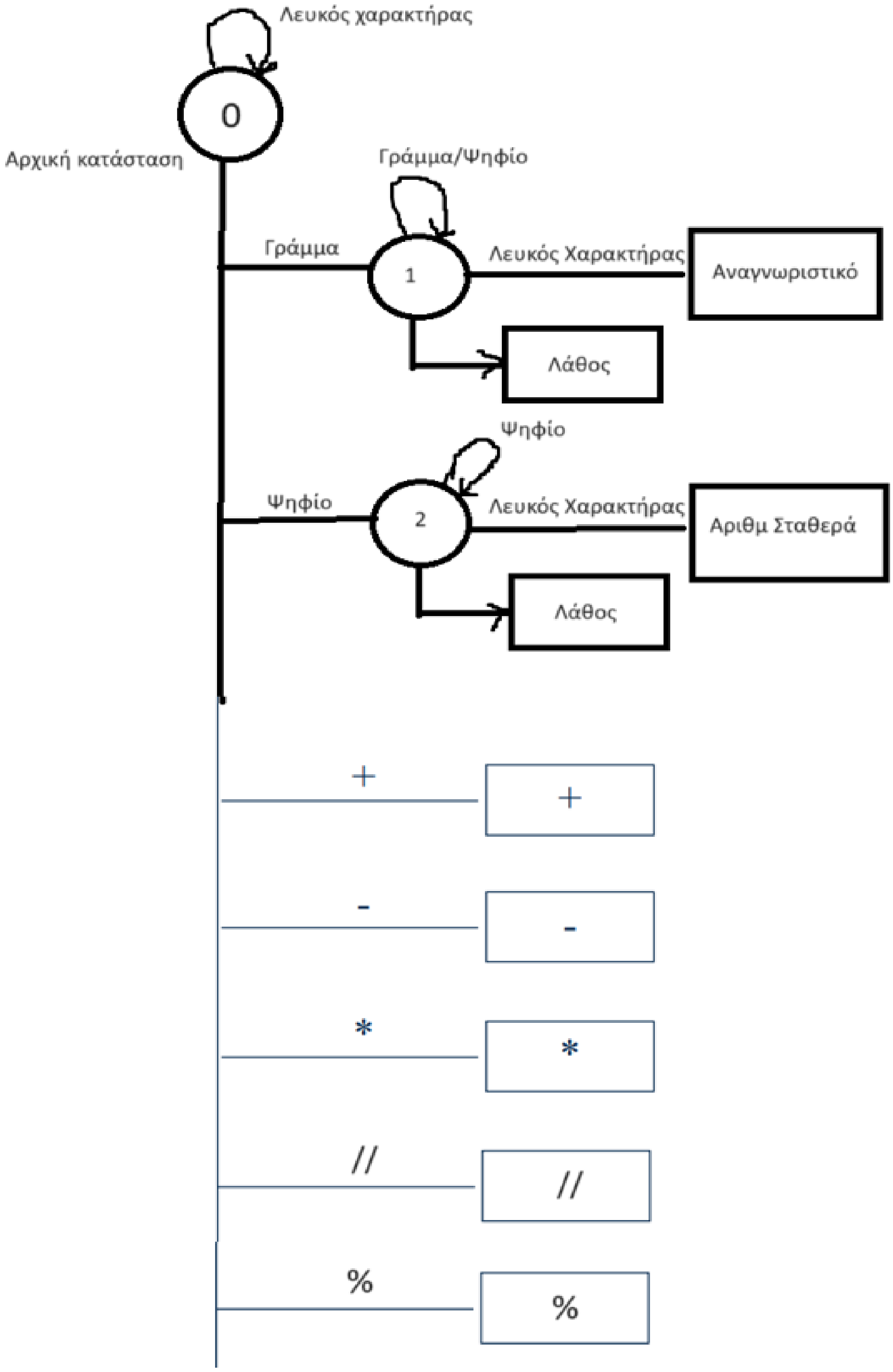

Figure 2

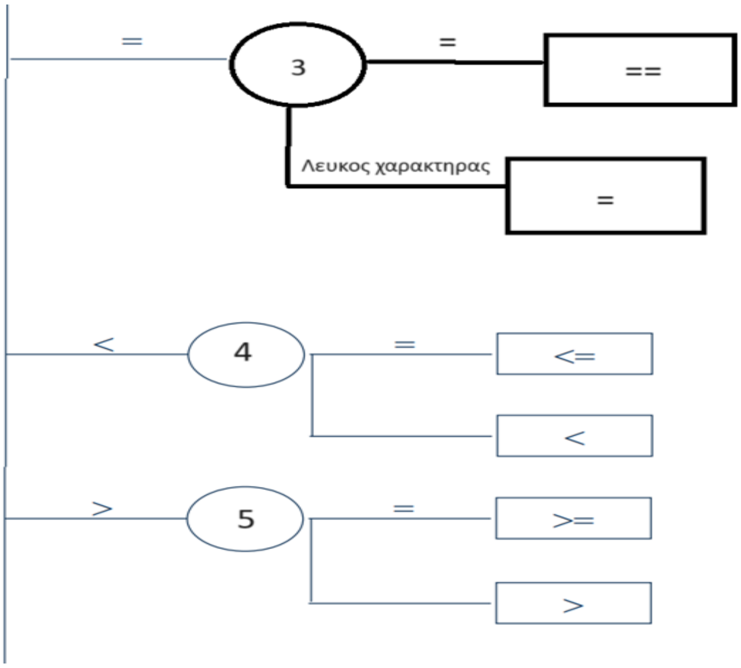

Figure 3

Also at this stage the check is made for the limit values given in the pronunciation. Finally, when an error is detected either by the automatic or by the limit values, the corresponding error message is printed and compilation is terminated.

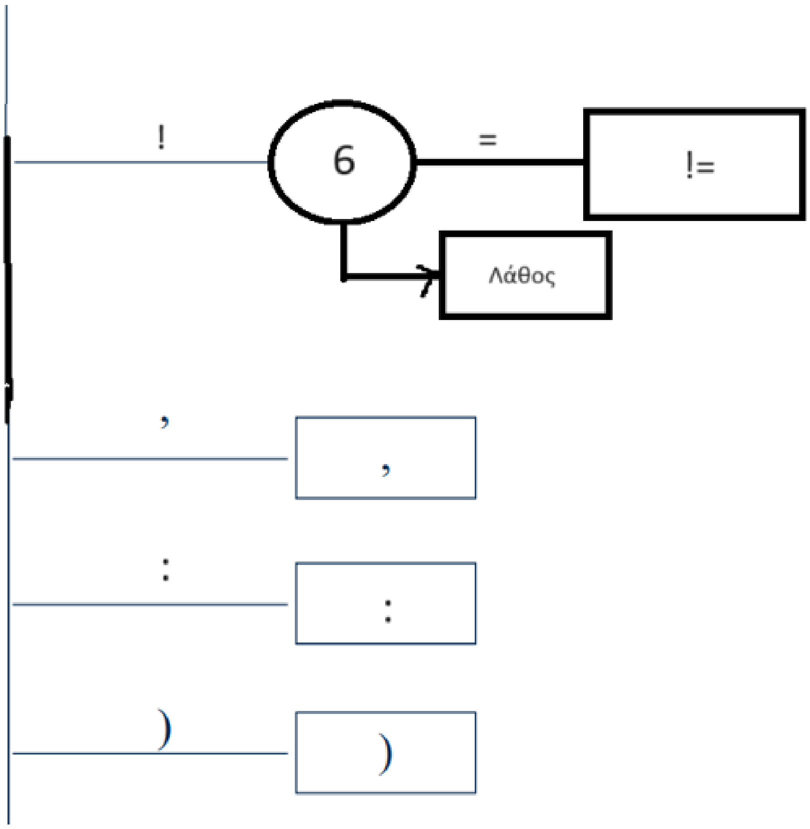

Figure 4

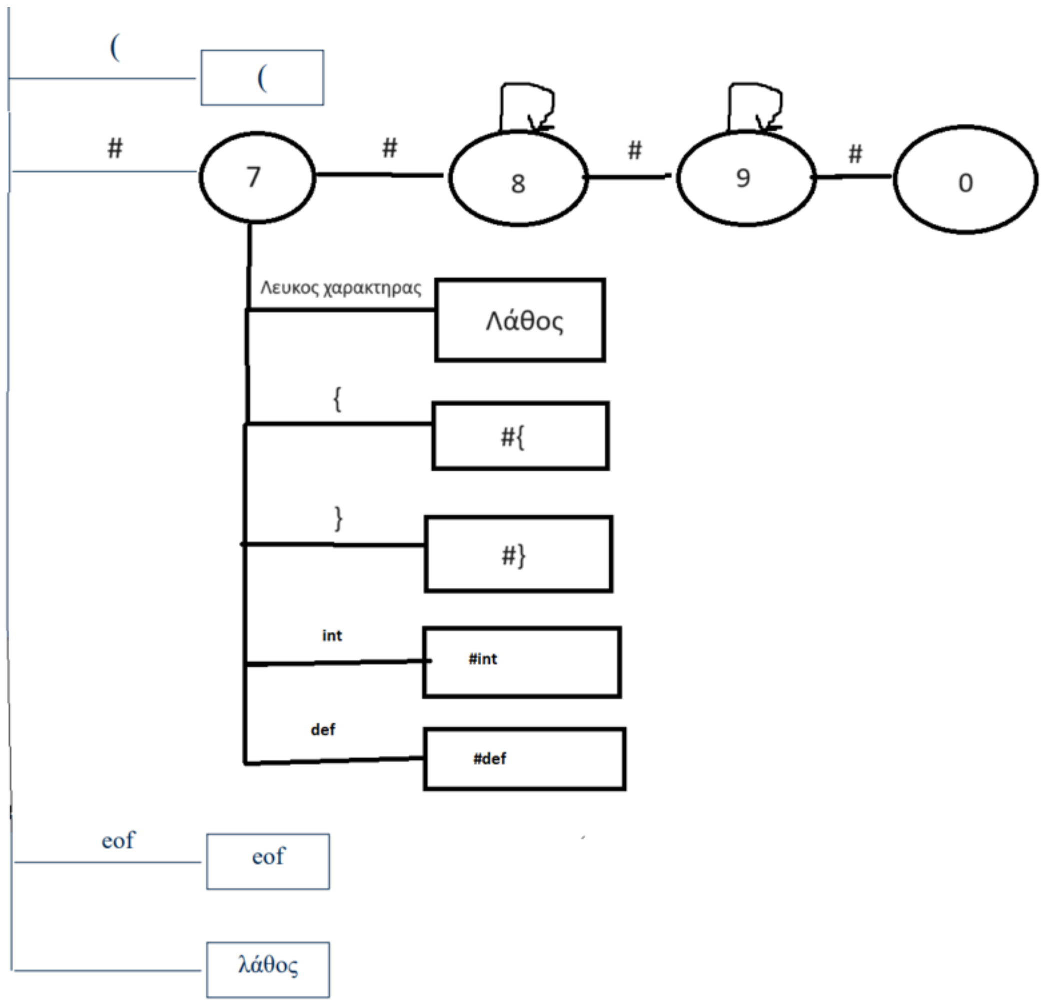

Figure 5

Machine explanation/code:

Situations

- **Status 0**: Original state. It reads the current character and decides the next state based on that character.
  - If the character is a blank, newline, carriage return, tab, or vertical tab, it remains in state 0.
  - If the character is a letter, it goes to state 1. If the
  - character is a digit, it goes to state 2. If the character is=,
  - goes to state 3. If the character is<, goes to state 4. If the
  - character is>, goes to state 5. If the character is!, goes to
  - state 6.
  -

- If the character is#, goes to state 7.
- If the character is+, returns the token["addtoken", line]. If the
- character is-, returns the token["subtoken", line]. If the character
- is*, returns the token["multitoken", line]. If the character is/, goes
- to state 14.
- If the character is%, returns the token["modtoken", line]. If the
- character is,, returns the token["commatoken", line].
- If the character is:, returns the token["anwkatwtoken", line].
- If the character is(, returns the token["leftpartoken", line].
- If the character is), returns the token["rightpartoken", line].
- For any other character, it displays an error message and stops execution.

- **Situation 1**: Collects words. If the character is a letter or digit, adds it to the word. Otherwise, it returns the appropriate token.
  - If the word exists incommandList, returns the token ["commandtoken", line, word].
  - If the word is a unique identifier (up to 30 characters), returns the token["anagnoristikotoken", line, word].
- **Situation 2**: Collects numbers. If the character is a digit, adds it to the number. Otherwise, it returns the token["numbertoken", line, int(number)]if the number is within limits.
- **Situation 3**: Checks if the next character is=.
  - If it is, it returns the token["isothtatoken", line]. Otherwise,
  - it returns the token["ana8eshtoken", line].
- **Status 4**: Checks if the next character is=.
  - If it is, it returns the token["mikistoken", line]. Otherwise, it
  - returns the token["mikroterotoken", line].
- **Status 5**: Checks if the next character is=.
  - If it is, it returns the token["megisotoken", line]. Otherwise, it
  - returns the token["megaluterotoken", line].
- **Status 6**: Checks if the next character is=.
  - If it is, it returns the token["diaforotoken", line]. Otherwise, it
  - displays an error message and stops execution.
- **Status 7**: Checks for comments and symbols{,}or recognize special tokens#i(forint) and#d(fordef).
- **Status 8**: Manages comments.
- **Situation 9-12**: Check for special words#intand#def.
- **Status 13**: Continues comments.
- **Status 14**: Checks for//(integer division).

If the function reaches the end of the file, it returns the token["EOFtoken", line].

4) Editorial analyst

Its grammar was used to implement the syntax analyzer

language cpy . Her grammar followscpy which also gives the exact description

of the language:

```
startRule
        :
          declarations
          def_function
      call_main_part
        ;

def_function
        :'def'ID'('id_list')' ':'
            '#{'
                Declarations
                (def_function)*
                 globals
              statements


            '#}'
        ;

declarations
        : (declaration_line)*
        ;

declaration_line
        :'#int'id_list
        ;

globals
       : (globals_line)*
        ;

globals_line
        :globalid_list
;

statement
        :simple_statement
        | structured_statement
        ;
```

```
statements
            :statement+
            ;

simple_statement
            :assignment_stat
            | print_stat
            | return_stat
            ;

structured_statement
            :if_stat
            | while_stat
            ;


assignment_stat
            : ID'='
                  (expression
                                    |'int' '(' 'input' '(' ')' ')'
                  )
            ;

print_stat
            :'print' '('expression')' ;


return_stat
            :'return'expression
            ;

if_stat

           :'if'condition':'
                     (statement
                     )
                ('elif'condition':'
                     (statement
                     )
                )
                ('else':'
                     (statement
           ;         )
                ) ?
while_stat
```

```
            :    whilecondition':'
                 '#{'
                      (statement
                      )
            ;    '#}'

id_list
            :      ID (','ID )*
            |
            ;

expression
            :    optional_sign term
                 (ADD_OP term)*
            ;

term
            :    factor
                 (MUL_OP factor)*
            ;



Factor
            :          INTEGER
            |          '(' expression ')'
            |          idtail ID
            ;


idtail
             :'('actual_par_list')'
             |
             ;

actual_par_list
             :          expression(','expression)*
             |
             ;


Optional_sign
             :        ADD_OP
             |
             ;

condition
             :bool_term(orbool_term)* ;
```

```
bool_term
        :bool_factor(andbool_factor)*
        ;

bool_factor
        :'not' '['condition']' |'['condition']'

         | expression REL_OP expression
        ;

call_main_part:
        '#def main'
         declarations
        statements
        ;
```

The parser in each case reads the parser an array containing the token, the line, and in some cases anything else that is useful. When an error is detected, the corresponding error message is printed and compilation is terminated.

Grammar implementation in python:

if_stat rule:

- • **Check if thetoken iscommandtokenand if the command isiftheelif**:
  - If so, it continues processing.
  - Otherwise, the function returns.
- **Condition Checking and Reading**:
  - It calls the functionisCondition(token)to check if the next token is a condition.
  - If the condition is valid, it saves the listsconditionTrueand conditionFalse.
  - If not, it displays an error message and terminates execution.
- **Backpatch and Read Next Token**:
  - Updates listsconditionTruewith the next quartet.
  - Reads the next token.
- **Check for:After theiftheelif**:
  - If the token isanwkatwtoken(ie:), continues.

- If not, it displays an error message and terminates execution.

- **Checking and Execution of Declaration**:

  - If the statement is valid (isStatement(token)), continues.
  - Creates a listifListwith the next four and adds one quartetjump.

  - Updates the listconditionFalsewith the next quartet.
  - Saves the file location and reads the next token.

- **Check for Next Command**:

  - If the next command iselif, retroactively calls herifState(token).
  - If the next command iselse, calls herelseState(token).
  - If it's any other command, it updates the listifListand returns the location of the file.

```python
def ifState(token):
    if token[0] == "commandtoken":
        if token[2] == "if" or token[2] == "elif":
            token = lex()
            cond = isCondition(token)
            if cond[0]:
                conditionTrue = cond[1]
                conditionFalse = cond[2]
                backpatch(conditionTrue, nextQuad())
                token = lex()
                if token[0] == "anwkatwtoken":
                    token = lex()
                    if isStatement(token):
                        ifList = makeList(nextQuad())
                        quadList.append(genQuad("jump","_","_","_"))
                        backpatch(conditionFalse, nextQuad())
                        seekIndex = f.tell()
                        token = lex()
                        if token[0] == "commandtoken":
                            if token[2] == "elif":
                                x = ifState(token)
                                backpatch(ifList,nextQuad())
                                return x
                            elif token[2] == "else":
                                x = elseState(token)
                                backpatch(ifList,nextQuad())
                                return x
                            else:
                                backpatch(ifList,nextQuad())
                                f.seek(seekIndex)
                                return True
                        else:
                            backpatch(ifList,nextQuad())
                            f.seek(seekIndex)
                            return True
                    else:
                        print ("Inside the 'if' at line:", token[1], "a statement was expected")
                        print("Compilation failed")
                        exit()
                else:
                    print ("After the 'if' at line:", token[1], "a ':' was expected")
                    print("Compilation failed")
                    exit()
            else:
                print ("After the 'if' at line:", token[1], "a condition was expected")
                print("Compilation failed")
                exit()
```

Figure 6

- **Check If Command Iselse**:
  - The function starts by verifying that the passed token is the commandelse.
  - If it is, it reads the next token with the functionlex().
- **Control For:After theelse**:
  - If the next token isanwkatwtoken(ie:), moves to the next line.
  - If not, it displays an error message and terminates execution.
- **Checking and Execution of Declaration**:
  - Reads the next token and checks if it is a valid statement using the functionisStatement(token).
  - If the statement is valid, it returnsTrue.
  - If not, it displays an error message and terminates execution.
- **Errors and Exceptions**:
  - If any problem is found (like missing:or invalid statement), displays an error message and terminates execution.

```
def elseState(token):
    if token[2] == "else":
        token = lex()
        if token[0] == "anwkatwtoken":
                token = lex()
                if isStatement(token):
                    return True
                else:
                    print ("Inside the 'else' at line:", token[1], "a statement was expected")
                    print("Compilation failed")
                    exit()
        else:
                print ("After the 'else' at line:", token[1], "a ':' was expected")
                print("Compilation failed")
                exit()
```

Figure 7

Condition rule:

- **Initialize Lists**:
  - The listsBtrueandBfalseare initialized as empty.

- **Check and Read Logical Term**:
  - It calls the functionisBoolTerm(token)to check if the nexttoken is a logical term.
  - If the boolean is valid, it stores the listsQ1 trueandQ1false.
  - If not, it displays an error message and terminates execution.
- **Initialize ListsBtrue and Bfalse**:
  - The listsBtrueandBfalseare initialized with the values ofQ1 trueand Q1falserespectively.
- **Repeat for the Doeror**:
  - Saves the file location and reads the next token.
  - If the token is the commandor, continues processing.
  - It calls the functionisBoolTerm(token)to check the next logic term after theor.
  - If the logical condition is not valid, it displays an error message and terminates execution.
  - Updates listsBtrueandBfalsewith the new prices.
- **Return Result**:
  - If a valid boolean is found, it returns a list of three values: [True, Btrue, Bfalse].
  - If not, it displays an error message and terminates execution.

```
def isCondition(token):
    Btrue = []
    Bfalse = []
    global seekIndex
    bterm1 = isBoolTerm(token)
    if bterm1[0]:
        Q1true = bterm1[1]
        Q1false = bterm1[2]
        Btrue = Q1true
        Bfalse = Q1false
        seekIndex = f.tell()
        token = lex()
        if token[0] == "commandtoken":
            while token[2]=="or":
                backpatch(Bfalse,nextQuad())
                seekIndex = f.tell()
                token = lex()
                bterm2 = isBoolTerm(token)
                if bterm2[0] == False:
                    print("After the 'or' at line:", token[1], "a boolean term was expected")
                    print("Compilation failed")
                    exit()
                else:
                    Q2true = bterm2[1]
                    Q2false = bterm2[2]
                    Btrue = mergeList(Btrue,Q2true)
                    Bfalse = Q2false
                    f.seek(seekIndex)
            return [True,Btrue,Bfalse]
        else:
            f.seek(seekIndex)
            return [True,Btrue,Bfalse]
    else:
        print("At line:", token[1],"a boolean term was expected")
        print("Compilation failed")
        exit()
```

Figure 8

boolTerm rule:

- **Initializing Variables**:
  - Initializes the listsQtrueandQfalseas blanks.
- **Control and Analysis of the First Logical Factor**:
  - It calls the functionisBoolFactor(token)to check the first logic factor.
  - If the first factor is valid, it stores the listsR1 trueand R1false.
  - If invalid, it displays an error message and terminates execution.
- **Processing of the Terms with the Operatorand**:
  - If the first factor is valid, it checks whether the operator follows and.
  - If there is theand, returns to the next logical factor.
  - It repeats this process until the operator no longer exists and.
  - Returns listsQtrueandQfalseafter processing.
- **Return of Results**:
  - Returns a list of three values: [True, Qtrue, Qfalse].
  - If the first logical factor is not valid, it displays an error message and terminates execution.

```python

def isBoolTerm(token):
    Qtrue = []
    Qfalse = []
    global seekIndex
    bfactor1 = isBoolFactor(token)
    if bfactor1[0]:
        R1true = bfactor1[1]
        R1false = bfactor1[2]
        Qtrue = R1true
        Qfalse = R1false
        seekIndex = f.tell()
        token = lex()
        if token[0] == "commandtoken":
            while token[2]=="and":
                backpatch(Qtrue,nextQuad())
                seekIndex = f.tell()
                token = lex()
                bfactor2 = isBoolFactor(token)
                if bfactor2[0]:
                    R2true = bfactor2[1]
                    R2false = bfactor2[2]
                    Qfalse = mergeList(Qfalse,R2false)
                    Qtrue = R2true
                    seekIndex = f.tell()
                    token = lex()
                    if token[0]!= "commandtoken":
                        break
                else:
                    print("After the 'and' at line:", token[1], "a boolean factor was expected")
                    print("Compilation failed")
                    exit()
            f.seek(seekIndex)
            return [True,Qtrue,Qfalse]
        else:
            f.seek(seekIndex)
            return [True,Qtrue,Qfalse]
    else:
        print("At line:", token[1],"a boolean factor was expected")
        print("Compilation failed")
        exit()
```

Figure 9

boolFactor rule:

- **Initializing Variables**:
    - Creates the listsRtrueandRfalseblanks.
- **Controlling the Press of the Firsttokens**:
    - If the first token is a command (itemcommandtoken), then it must be an expression or a condition.
        - o If it is an expression, it checks for the comparison operator and a second expression.
        - o If it is condition with thenot, parses the corresponding condition and returns the opposite setsRtrueandRfalse.
    - If the first token is not a command, then it can be either an expression or a condition.

```
def isBoolFactor(token):
    Rfalse = []
    if token[0] == "commandtoken":
        expr1 = isExpression(token)
        if expr1[0]:
            E1place = expr1[1]
            token = lex()
            if token[0] in relOpList:
                if token[0] == "isothtatoken":
                    relOp = "="
                elif token[0] == "mikroterotoken":
                    relOp = "<"
                elif token[0] == "megaluterotoken":
                    relOp = ">"
                elif token[0] == "mikisotoken":
                    relOp = "<="
                elif token[0] == "megisotoken":
                    relOp = ">="
                else:
                    relOp = "!="
                token = lex()
                expr2 = isExpression(token)
                if expr2[0]:
                    E2place = expr2[1]
                    Rtrue = makeList(nextQuad())
                    quadList.append(genQuad(relOp, E1place,E2place,"_"))
                    Rfalse = makeList(nextQuad())
                    quadList.append(genQuad("jump","_","_","_"))
                    return [True,Rtrue,Rfalse]
                else:
                    print("At line", token[1], "an expression was expected after a relationship opperand")
                    print("Compilation failed")
                    exit()
```

Figure 10

- **Apply Comparison Operator**:
    - If the next token is a comparison operator, then it performs the corresponding comparison with the next expression.
- **Return Results**:
    - Returns a list of three values: [True, Rtrue, Rfalse] if the analysis is successful, otherwise returns [False].

```python
        elif token[2] == "not":
            token = lex()
            cond = isCondition(token)
            if cond[0]:
                Btrue = cond[1]
                Bfalse = cond[2]
                Rtrue = Bfalse
                Rfalse = Btrue
                return [True,Rtrue,Rfalse]
            else:
                print("At line:", token[1],"a boolean factor was expected after 'not")
                print("Compilation failed")
                exit()
        else:
            return [False]
    else:
        expr1 = isExpression(token)
        if expr1[0]:
            E1place = expr1[1]
            token = lex()
            if token[0] in relOpList:
                if token[0] == "isothtatoken":
                    relOp = "="
                elif token[0] == "mikroterotoken":
                    relOp = "<"
                elif token[0] == "megaluterotoken":
                    relOp = ">"
                elif token[0] == "mikisotoken":
                    relOp = "<="
                elif token[0] == "megisotoken":
                    relOp = ">="
                else:
                    relOp = "!="
                token = lex()
                expr2 = isExpression(token)
                if expr2[0]:
                    E2place = expr2[1]
                    Rtrue = makeList(nextQuad())
                    quadList.append(genQuad(relOp, E1place,E2place,"_"))
                    Rfalse = makeList(nextQuad())
                    quadList.append(genQuad("jump","_","_","_"))
                    return [True,Rtrue,Rfalse]
```

Figure 11

```python
                    quadList.append(genQuad("jump","_","_","_"))
                    return [True,Rtrue,Rfalse]
                else:
                    print("At line", token[1], "an expression was expected after a relationship opperand")
                    print("Compilation failed")
                    exit()

    else:
        cond = isCondition(token)
        if cond[0]:
            Btrue = cond[1]
            Bfalse = cond[2]
            Rtrue = Btrue
            Rfalse = Bfalse
            return [True,Rtrue,Rfalse]
        else:
            return [False]
```

Figure 12

Rule expression:

- **Initializing Variables**:
  - Initially, variables such asEplace,seekIndex,negSign.
- **Optional Premise Check**:
  - If the current token has an optional token (addtoken or subtoken), it parses the next token.
  - If the token is a subtoken, then the variablenegSigndefined asTrue.

```python
def isExpression(token):
    Eplace = 0
    global seekIndex
    global pos
    global scope
    seekIndex = f.tell()
    negSign = False
    if isOptionalSign(token):
        if token[0] == "subtoken":
            negSign = True
        token = lex()
    term1 = isTerm(token)
    if term1[0]:
        T1place = term1[1]
        if negSign:
            y = newTemp()
            pos=pos+4
            newPos=pos
            newScope=scope-1
            recordStructure.addNewEntity(scopeIndex=newScope,entityName=y,entity_type='TemporaryVariable', offset=newPos)
            quadList.append(genQuad("-",0,T1place,y))
            T1place = y
        seekIndex = f.tell()
        token = lex()
```

Figure 13

- **Analysis of Terms**:
  - The numerical terms of the expression are analyzed.
  - If an optional "-" sign is present, reverse orientation is applied.
  - For each addtoken or subtoken that follows, a new term is parsed and the appropriate value assignment is made.
- **Return Results**:
  - Returns True and the place of the result (Eplace) if the analysis is successful.
  - Otherwise, it returns False.

```python
        while token[0] == "addtoken" or token[0] == "subtoken":
            if token[0] == "addtoken":
                operand = "+"
            else:
                operand = "-"
            token = lex()
            term2 = isTerm(token)
            if term2[0]:
                T2place = term2[1]
                w = newTemp()
                pos=pos+4
                newPos=pos
                newScope=scope-1
                recordStructure.addNewEntity(scopeIndex=newScope,entityName=w,entity_type='TemporaryVariable', offset=newPos)
                quadList.append(genQuad(operand,T1place,T2place,w))
                T1place = w
                seekIndex = f.tell()
                token = lex()
            else:
                print("At line", token[1], "a term was expected after an add or sub opperand. Instead found:", token[0])
                print("Compilation failed")
                exit()
        f.seek(seekIndex)
        Eplace = T1place
        return [True,Eplace]
    else:
        f.seek(seekIndex)
        return [False]
```

Figure 14

optionalSign rule:

Checking the token:

- If the current token is addtoken or subtoken, then it is considered an optional token and the function returns True.
- In any other case, it returns False.

```python
def isOptionalSign(token):
    global seekIndex
    if token[0] == "addtoken" or token[0] == "subtoken":
        return True
    else:
        return False
```

Figure 15

Term rule:

- **Initializing Variables**:
  - Variables such as n are initializedTplace,seekIndex.
- **Factor Analysis**:
  - The numerical factors of the term are analyzed.
  - For each subsequent multiplication or division, a new factor is analyzed and assigned the appropriate values.
- **Return Results**:
  - Returns True and the position of the result (Tplace) if the analysis is successful.
  - Otherwise, it returns False.

```python
def isTerm(token):
    Tplace = 0
    global seekIndex
    global pos
    global scope
    factor1 = isFactor(token)
    if factor1[0]:
        F1place = factor1[1]
        seekIndex = f.tell()
        token = lex()
        while token[0] == "multoken" or token[0] == "modtoken" or token[0] == "divtoken":
            if token[0] == "multoken":
                operand = "*"
            elif token[0] == "divtoken":
                operand = "/"
            else:
                operand = "mod"
            token = lex()
            factor2 = isFactor(token)
            if factor2[0]:
                F2place = factor2[1]
                w = newTemp()
                pos=pos+4
                newPos=pos
                newScope=scope-1
                recordStructure.addNewEntity(scopeIndex=newScope,entityName=w,entity_type='TemporaryVariable', offset=newPos)
                quadList.append(genQuad(operand,F1place,F2place,w))
                F1place = w
                seekIndex = f.tell()
                token = lex()
            else:
                print("At line", token[1], "a factor was expected after a multiplication or division opperand. Instead found: ",token[0])
                print("Compilation failed")
                exit()
        f.seek(seekIndex)
        Tplace = F1place
        return [True,Tplace]
    else:
        return [False]
```

Figure 16

factor rule:

- **Controltokens**:
  - If the current token is a numbertoken, the function returns its value and True.

- If the token is an identifier (anagnoristikotoken), it is checked if the token's tail follows.
- If the token is a left parenthesis (leftpartoken), the following expression is analyzed and the corresponding right parenthesis is expected.

- **Return Results**:

    - Returns True and the factor's position (Fplace) if the analysis is successful.
    - Otherwise, it returns False.

```python
def isFactor(token):
    Fplace = 0
    global seekIndex
    global pos
    global scope
    if token[0] == "numbertoken":
        Fplace = token[2]
        return [True,Fplace]
    elif token[0] == "anagnoristikotoken":
        idName = token[2]
        seekIndex = f.tell()
        tempSeekIndex = seekIndex
        token = lex()
        if idTail(token):
            w = newTemp()
            pos=pos+4
            newPos=pos
            newScope=scope-1
            recordStructure.addNewEntity(scopeIndex=newScope,entityName=w,entity_type='TemporaryVariable', offset=newPos)
            quadList.append(genQuad("par",w,"ret","_"))
            quadList.append(genQuad("call",idName,"_","_"))
            return [True,w]
        else:
            IDplace = idName
            Fplace = IDplace
            f.seek(tempSeekIndex)
            return [True,Fplace]
    elif token[0] == "leftpartoken":
        token = lex()
        expr = isExpression(token)
        if expr[0]:
            Eplace = expr[1]
            Fplace = Eplace
            token = lex()
            if token[0] == "rightpartoken":
                return [True,Fplace]
            else:
                print("At line", token[1], "a ')' was expected")
                print("Compilation failed")
                exit()
        else:
            print("At line", token[1], "an expression was expected")
            print("Compilation failed")
            exit()
    else:
        return [False]
```

Figure 17

idTail rule:

- **ID Check**: If the current token is a left parenthesis (leftpartoken), then checks if a list of parameters (parList) follows and then expects a right parenthesis (rightpartoken).

• **Return Result**: Returns True if the queue is syntactically correct, that is, if there are left and right parentheses after a list of parameters. Otherwise, it returns False.

```python
def idTail(token):
    if token[0] == "leftpartoken":
        token = lex()
        if parList(token):
            token = lex()
            if token[0] == "rightpartoken":
                return True
            else:
                print("At line", token[1], "a ')' was expected")
                print("Compilation failed")
                exit()
        else:
            return False
    else:
        return False
```

Figure 18

parList rule:

• **Expression Analysis**: First checks if an expression exists. If so, then it stores the result of the expression in a variable and adds the parameter to the stack with the appropriate type (cvfor the guest Department). It also notes that at least one parameter exists.

• **Repetition**: If there is a parameter, the function repeats the process for each subsequent parameter that follows the separator,.

• **Return Result**: Returns True if there are any parameters in the list or False if there are no parameters.

```
def parList(token):
    global seekIndex
    global isAtLeastOnePar
    global pos
    global scope
    expr1 = isExpression(token)
    if expr1[0]:
        E1place = expr1[1]
        a = E1place
        pos=pos+4
        newPos=pos
        newScope=scope-1
        recordStructure.addNewEntity(scopeIndex=newScope,entityName=a,entity_type='Variable', offset=newPos)
        quadList.append(genQuad("par",a,"CV","_"))
        isAtLeastOnePar = True
        seekIndex = f.tell()
        token=lex()
        while token[0] == "commatoken":
            seekIndex = f.tell()
            token = lex()
            expr2 = isExpression(token)
            if expr2[0]:
                E2place = expr2[1]
                b = E2place
                pos=pos+4
                newPos=pos
                newScope=scope-1
                recordStructure.addNewEntity(scopeIndex=newScope,entityName=b,entity_type='Variable', offset=newPos)
                quadList.append(genQuad("par",b,"CV","_"))
                seekIndex = f.tell()
                token=lex()
                continue
            else:
                print("At line", token[1], "an expression was expected after the ','")
                print("Compilation failed")
                exit()
        f.seek(seekIndex)
        return isAtLeastOnePar
    else:
        f.seek(seekIndex)
        return True
```

Figure 19

Statement rule:

1. First it checks if the statement is a simple statement (simple statement) with the using the functionisSimpleStatement(token).
2. If the statement is not simple, it checks if it is a structured statement using the functionisStructuredStatement(token).

If control in either case returnsTrue, then the function also returnsTrue, indicating that the input is a statement. Otherwise, it returnsFalse.

```
def isStatement(token):
    if isSimpleStatement(token):
        return True
    elif isStructuredStatement(token):
        return True
    else:
        return False
```

Figure 20

simple_statement rule:

1. It first checks if the statement is an assignment using the function isAssignmentStat(token).
2. If the statement is not an assignment, it checks if it is a print statement using the functionisPrintStat(token).
3. If the statement is neither an assignment nor a print statement, it checks whether it is a return statement using the function isReturnStat(token).

If the control in any of the above cases returnsTrue, then the function also returnsTrue, indicating that the input is a simple statement. Otherwise, it returnsFalse.

```python
def isSimpleStatement(token):
    if isAssignmentStat(token):
        return True
    elif isPrintStat(token):
        return True
    elif isReturnStat(token):
        return True
    else:
        return False
```

Figure 21

assignment_stat rule:

• Check for ID (identifier): It first checks if the statement starts with an identifier, which is recognized by the "anagnoristikotoken" token. If not, the statement is not an assignment and the function returnsFalse.

• Check for the assignment operator: If the firsttoken is an identifier, checks if the assignment operator "=" follows the token "ana8eshtoken". If not, the statement is not an assignment and the function returnsFalse.

• Check for expression: If the previous checks pass, the function checks if an expression follows the assignment operator. This is done by calling the functionisExpression(token). If a valid expression exists, a quadruple is created for the assignment using the function genQuad(":=", Eplace, "_", idPlace)and the function returnsTrue. If no valid expression, an error message is displayed and the function returnsFalse.

```python
def isAssignmentStat(token):
    if token[0] == "anagnoristikotoken":
        idPlace = token[2]
        token = lex()
        if token[0] == "ana8eshtoken":
            token = lex()
            expr = isExpression(token)
            if expr[0]:
                Eplace = expr[1]
                quadList.append(genQuad(":=", Eplace, "_", idPlace))
                return True
            elif token[0] == "commandtoken":
                if token[2] == "int":
                    token = lex()
                    if token[0] == "leftpartoken":
                        token = lex()
                        if token[0] == "commandtoken":
                            if token[2] == "input":
                                token = lex()
                                if token[0] == "leftpartoken":
                                    token = lex()
                                    if token[0] == "rightpartoken":
                                        token = lex()
                                        if token[0] == "rightpartoken":
                                            quadList.append(genQuad("inp",idPlace,"_","_"))
                                            return True
                                        else:
                                            print("At line", token[1], "a ')' was expected")
                                            print("Compilation failed")
                                            exit()
                                    else:
                                        print("At line", token[1], "a ')' was expected")
                                        print("Compilation failed")
                                        exit()
                                else:
                                    print("At line", token[1], "a '(' was expected")
                                    print("Compilation failed")
                                    exit()
                            else:
                                return False
                        else:
                            return False
                    else:
                        print("At line", token[1], "a '(' was expected")
                        print("Compilation failed")
                        exit()
                else:
                    return False
```

Figure 22

• Additional Check for 'int(input())': If the expression is false and the next token is the statement "int", the function performs an additional check to confirm that the statement is of the form "int(input())". If this is true, a triple is created for the input using the function genQuad("inp", idPlace, "_", "_")and the function returnsTrue. If not is of the expected form

```python
            else:
                print("At line", token[1], "either an expression or 'int(input()) was expected after the '='")
                print("Compilation failed")
                exit()
        else:
            return False
    else:
        return False
```

Figure 23

return_stat rule:

• Check for return command: First checks if the firsttoken is the "return" command, which is identified by the "commandtoken" token. If it is not, the function returnsFalse.

• Check for expression: If the firsttoken is the "return" command, then it checks if an expression follows. This is done by calling the function isExpression(token). If a valid expression exists, a quadruple is created to return using the functiongenQuad("ret", Eplace, "_", "_")and the function returnsTrue. If there is no valid expression, an error message is displayed and the function returnsFalse.

```
def isReturnStat(token):
    if token[0] == "commandtoken":
        if token[2] == "return":
            token = lex()
            expr = isExpression(token)
            if expr[0]:
                Eplace = expr[1]
                quadList.append(genQuad("ret", Eplace, "_", "_"))
                return True
            else:
                print("At line", token[1], "an expression was expected after the 'return' command")
                print("Compilation failed")
                exit()
        else:
            return False
    else:
        return False
```

Figure 24

structure_statement rule:

• Check for statementif: Calls the functionifState(token)to check if token represents an if statement. If so, it returnsTrue.

• Check for statementwhile: Calls the functionwhileState(token)to check if the token represents a while statement. If so, it returnsTrue.

• ReturnFalse: If the token does not correspond to either an if statement or a while statement, it returnsFalse.

```
def isStructuredStatement(token):

    if ifState(token):
        return True
    elif whileState(token):
        return True
    else:
        return False
```

Figure 25

while_stat rule:

• Check if thetoken represents the statementwhile: It first checks if the token is a valid command (command token) and if it matches the "while" keyword. If not, it returnsFalse.

• Analysis of her bodywhile loop: After checking that the statement begins with the "while" keyword, it continues checking in the following order:

- It checks the condition of the while loop using the function isCondition(token).
- If the condition is valid, then it checks if it is followed by the correct '{' symbol with the functionlex().
- Next is checking for the statements inside the body of the while loop using the functionstatements(token).
- Checks whether the body of the while loop is closed with the appropriate '}' symbol.
- If all of the above is valid, it returnsTrue.

• ReturnFalse: If the token does not represent a while statement, then it returnsFalse.

```python
def whileState(token):
    if token[0] == "commandtoken":
        if token[2] == "while":
            condQuad = nextQuad()
            token = lex()
            cond = isCondition(token)
            if cond[0]:
                conditionTrue = cond[1]
                conditionFalse = cond[2]
                backpatch(conditionTrue, nextQuad())
                token = lex()
                if token[0] == "anwkatwtoken":
                    token = lex()
                    if token[0] == "anoigmatoken":
                        token = lex()
                        if statements(token):
                            token = lex()
                            if token[0] == "kleisimotoken":

                                quadList.append(genQuad("jump","_","_",condQuad))
                                backpatch(conditionFalse,nextQuad())
                                return True
                            else:
                                print("At line", token[1] ,"a '#}' was expected to close a '#{ found earlier for the while loop")
                                print("Compilation failed")
                                exit()
                        else:
                            print("At line", token[1], "a statement was expected after the ':'")
                            print("Compilation failed")
                            exit()
                    else:
                        print("At line", token[1], "a '#{' was expected after the 'while'")
                        print("Compilation failed")
                        exit()
                else:
                    print("At line", token[1], "a ':' was expected")
                    print("Compilation failed")
                    exit()
            else:
                print("At line", token[1], "a condition was expected after the 'while' command")
                print("Compilation failed")
                exit()
        else:
            return False
    else:
        return False
```

Figure 26

startRule rule:

- **Download nexttokens**: First, it calls the functionlex()to receive the first program token.

- **Create a new scope level**: Adds a new level of scope to the record structure for variables and functions.

- **Check statements**: Checks if there are declarations in the program using the functiondeclarations(token). If the declaration process is successful, it continues, otherwise it terminates the compilation.

- **Checking and calling functions**: Checks and calls the function isDefFunctions(token)for each subsequent onetoken until there are no more function declarations in the program.

- **Checking and calling the main section (main)**: Checks if the main part of the program with the function existscallMainPart(token). If all are valid, it adds the corresponding trailing quaternions (halt,end_block), displays the final quad and returnsTrue. If the main section is not found, the program terminates with an error.

```python
def startRule():
    token=lex()
    recordStructure.addNewScope()
    if declarations(token):
        token=lex()
        while isDefFunctions(token):
            token=lex()
        if callMainPart(token):
            quadList.append(genQuad("halt","_","_","_"))
            quadList.append(genQuad("end_block","main","_","_"))
            readQuadList(quadList[-1])
            return True
        else:
            exit()
    else:
        print("At line", token[1]," declarations was expected")
        print("Compilation failed")
        exit()
```

Figure 27

print_stat rule:

1. **Command type check**: First, it checks if the current token is a command. If it isn't, it returnsFalse.

2.**Print order check**: If the current token is a print command (print), continues checking, otherwise returnsFalse.
3.**Left bracket control**: Waits for the next token to be the left bracket ((). If it is not, it displays an error message and terminates the compilation.
4.**Print expression control**: Expects an expression after the left bracket. If an expression exists, it continues checking, otherwise it displays an error message and terminates compilation.
5.**Right bracket control**: Expects the right bracket ()) after the expression. If this is not the case, it displays an error message and terminates the compilation.

If all steps pass successfully, it adds a block of appropriate print command (out) in the final list of quads and returnsTrue.

```
def isPrintStat(token):
    global seekIndex
    if token[0] == "commandtoken":
        if token[2] == "print":
            token = lex()
            if token[0] == "leftpartoken":
                token = lex()
                expr = isExpression(token)
                if expr[0]:
                    Eplace = expr[1]
                    token = lex()
                    if token[0] == "rightpartoken":
                        quadList.append(genQuad("out",Eplace,"_","_"))
                        seekIndex = f.tell()
                        return True
                    else:
                        print("At line", token[1]," a ')' was expected")
                        print("Compilation failed")
                        exit()
                else:
                    print("At line", token[1], "an expression was expected after the 'print' command")
                    print("Compilation failed")
                    exit()
            else:
                print("At line", token[1]," a '(' was expected after the 'print' command")
                print("Compilation failed")
                exit()
        else:
            return False
    else:
        return False
```

Figure 28

id_List rule:

• **Initialize variables**: Initially, two variables are defined, hseekIndexand theisAtLeastOneID.

• **ID check**: If the first token is an identifier, it is assumed that there is at least one identifier in the list and its position is recorded in the variable seekIndex. The identifier is added to the listnames.

• **Repeat for more IDs**: If a comma follows, checks if another identifier follows. If so, it adds it to the listnamesand records its position with theseekIndex. It repeats this process until the list of IDs is finished.

- **Return results**: Returns a list containing a logical boolean (isAtLeastOneID) which indicates if there is at least one identifier and a list (names) with the identifiers found.

```python
def isIdList(token):
    global seekIndex
    global isAtLeastOneID
    global pos
    global scope
    names =[]
    if token[0]=="anagnoristikotoken":
        isAtLeastOneID = True
        seekIndex = f.tell()
        names.append(token[2])
        token=lex()
        newScope=scope-1
        while token[0]=="commatoken":
            token=lex()
            if token[0]=="anagnoristikotoken":
                names.append(token[2])
                seekIndex = f.tell()
                token=lex()
            else:
                print("At line", token[1]," an id was expected")
                print("Compilation failed")
                exit()
        f.seek(seekIndex)
        for x in names:
            pos=pos+4
            newPos=pos
            recordStructure.addNewEntity(scopeIndex=newScope,entityName=x,entity_type='Variable', offset=newPos)
        return [isAtLeastOneID,names]
    else:
        f.seek(seekIndex)
        return [False,names]
```

Figure 29

Rule statements:

- **Check statement**: First, it checks if the first token represents a statement. If not, it returnsFalse.

- **Repeat for more statements**: If the first token represents a statement, then it iterates to check if there are more statements in the sequence. If not, it returnsTrue.

- **Return result**: ReturnsTrueif all statements are valid, otherwiseFalse.

```python
def statements(token):
    global seekIndex
    if isStatement(token) == False:
        f.seek(seekIndex)
        return False
    f.seek(seekIndex)
    token = lex()
    while isStatement(token):
        seekIndex = f.tell()
        f.seek(seekIndex)
        token=lex()

    f.seek(seekIndex)
    return True
```

Figure 30

Rule declarations:

- **Check statement**: First, it checks if the first token represents a statement. If there is at least one statement, the variable isAtLeastOneDeclarationis set toTrue.

- **Repeat for more statements**: If there is at least one statement and the variableisAtLeastOneDeclarationisTrue, then an iteration is performed to test the remaining statements as well.

- **Return result**: Returns the value of the variable isAtLeastOneDeclaration, which indicates whether there is at least one valid statement or not.

```python
def declarations(token):
    global seekIndex
    global isAtLeastOneDeclaration
    seekIndex = f.tell()
    isAtLeastOneDeclaration = isDeclaration(token)
    token=lex()
    while isDeclaration(token) and isAtLeastOneDeclaration:
        seekIndex = f.tell()
        token = lex()
    f.seek(seekIndex)
    return isAtLeastOneDeclaration
```

Figure 31

Rule globals:

- **Check statement**: First, it checks if the first token represents a declaration for a global variable. If there is at least one statement, the variable isAtLeastOneGlobalis set toTrue.

- **Repeat for more statements**: If there is at least one statement and the variableisAtLeastOneGlobalisTrue, then an iteration is performed to check the remaining declarations for global variables as well.

- **Return result**: Returns the value of the variable isAtLeastOneGlobal, which indicates whether there is at least one valid declaration for a global variable or not.

```
def globalVar(token):
    global seekIndex
    global isAtLeastOneGlobal
    seekIndex = f.tell()
    isAtLeastOneGlobal = isGlobal(token)
    token=lex()
    while isGlobal(token):
        seekIndex = f.tell()
        token=lex()
    f.seek(seekIndex)
    return isAtLeastOneGlobal
```

Figure 32

Declaration rule:

- **Statement type check**: First, it checks if the first token is one intdeftoken, which indicates a type variable declarationint.

- **Analysis of itidList**: If the declaration is of typeint, continues checking with the functionisIdListto check if it is followed by a valid list identifiers.

- **Return result**: ReturnsTrueif the statement is valid, that is if it starts with intdeftokenand followed by a valid list of identifiers, otherwise it returnsFalse.

```python
def isDeclaration(token):
    if token[0]=="intdeftoken":
        token=lex()
        idList =isIdList(token)
        if idList[0]:
            return True
        else:
            print("At line", token[1], "an id was expected ")
            print("Compilation failed")
            exit()
    else:
        return False
```

Figure 33

Global rule:

- **Check declaration type**: First, it checks if the first token is one commandtokenand if the command isglobal.

- **Analysis of itidList**: If the command is of typeglobal, continues checking with the functionisIdListto check if it is followed by a valid list identifiers.

- **Return result**: ReturnsTrueif the command is valid, that is if it starts with globaland followed by a valid list of identifiers, otherwise returnsFalse.

```python
def isGlobal(token):
    if token[0]=="commandtoken":
        if token[2]=="global":
            token=lex()
            idList =isIdList(token)
            if idList[0]:
                return True
            else:
                print("At line", token[1], "an id was expected ")
                print("Compilation failed")
                exit()
        else:
            return False
    else:
        return False
```

Figure 34

def_function rule:

- **Check the first onetokens:**
  - It starts by checking if the first token is onecommandtokenand if the command isdef. This indicates that a function declaration is found.

- **Parsing the function name**:
  - If the current token isdef, then we expect the function name to follow (aanagnoristikotoken). This name is stored in the variableq.
  - Then we wait for oneleftpartoken, indicating the beginning of the function's argument list.
  - The following is the parsing of the function's argument list, which is done by calling the functionisIdList.
  - Then we store the length of the function frame (which is calculated based on the number of arguments) in the variable framelen.
  - Additionally, we create a new record in the program's data structure to store the function, including its arguments.

```python
def isDefFunctions(token):
    global scope
    global pos
    if token[0] == "commandtoken":
        if token[2] == "def":
            token=lex()
            if token[0] == "anagnoristikotoken":
                lastBlock.append(token[2])
                q = token[2]
                pos=pos+4
                newPos=pos
                newScope=scope-1
                token=lex()
                if token[0] == "leftpartoken":
                    token = lex()
                    idList =isIdList(token)
                    if idList[0]:
                        framelen=newPos
                        recordStructure.addNewEntity(scopeIndex=newScope,entityName=q,entity_type='Function', framelength=newPos,arguments=idList[1])
                        token=lex()
                        if token[0] == "rightpartoken":
                            token = lex()
                            if token[0] == "anwkatwtoken":
                                recordStructure.addNewScope()
                                token = lex()
                                if token[0] == "anoigmatoken":
                                    token = lex()
                                    if declarations(token):
                                        token = lex()
                                    hasFunction = False
                                    while True:
                                        if isDefFunctions(token):
                                            hasFunction = True
                                            token = lex()
                                            continue
                                        else:
                                            if hasFunction == False:
                                                quadList.append(emptyList())
                                                indexList.append(len(quadList)-1)
                                            break
```

Figure 35

```python
            if statements(token):
                token = lex()

                if token[0] == "kleisimotoken":
                    quadList.append(genQuad("end_block",lastBlock[-1],"_","_"))

                    pos=newPos
                    if len(lastBlock) == 1 and len(indexList) == 0:
                        lastBlock.pop()
                        readQuadList(quadList[-1])
                        return True
                    if len(indexList)>0:
                        quadList[indexList[-1]] = ["begin_block",lastBlock.pop(),"_","_"]

                        indexList.pop()
                        readQuadList(quadList[-1])
                        if len(indexList) == 0 and len(lastBlock)>0:

                            quadList.append(genQuad("begin_block",lastBlock[-1],"_","_"))
                    else:

                        quadList.append(genQuad("begin_block",lastBlock[-1],"_","_"))

                    return True
                else:
                    print ("At line", token[1], "a '#}' was expected to close a '#{' found earlier")
                    print("Compilation failed")
                    exit()
            else:
                print ("At line", token[1], "a statement was expected")
                print("Compilation failed")
                exit()
        else:
            print ("At line", token[1], "a '#{' was expected")
            print("Compilation failed")
            exit()
    else:
        print ("At line", token[1], "a ':' was expected")
        print("Compilation failed")
        exit()
else:
    print ("At line", token[1], "a ')' was expected")
    print("Compilation failed")
    exit()
```

Figure 36

- **Analysis of the body of the function**:
    - Following is the analysis of the function body.
    - Statements are parsed using the functiondeclarations.
    - Check for other built-in functions using it isDefFunctions.
    - The following is the analysis of the variables (global variables) using it globalVar.
    - Finally, the analysis of the commands of the body of the function follows use of itstatements.

- **Completion of the analysis**:
    - Completes parsing of the function body.
    - Adds the necessary final notebooks (genQuad) for the output.

```
                            else:
                                print ("At line", token[1], "a ')' was expected")
                                print("Compilation failed")
                                exit()
                        else:
                            print ("At line", token[1], "an id was expected")
                            print("Compilation failed")
                            exit()
                    else:
                        print ("At line", token[1], "a '(' was expected")
                        print("Compilation failed")
                        exit()
                else:
                    print ("At line", token[1], "an id was expected")
                    print("Compilation failed")
                    exit()
            else:
                return False
        else:
            return False
```

Figure 37

- **Addition of the final quatrains**:
  - Endnotes are added to terminate the function.
  - If there is only one function layer and no built-in functions (isDefFunctions), then the command is takenhaltfor termination of the program.
  - The final notebook is enteredend_blockto mark its end body of the function.

- **Return result**:
  - ReturnsTrueafter the function analysis has completed successfully.
  - If any of the expected commands are missing, the program prints an error message and terminates.

call_main_part rule:

- **Check function definitionmain**: First, it checks if the first token is defitokenand whether the function has a namemain.

- **Create function record**: If the functionmainit is valid, adds a record for the functionmainin the data structure recordStructure. This record includes the name, framelength, which in your case is the size of the main data structurescope(framelen), as well as the argument list (empty for the main).

- **Creation of the top four**: Adds a special quad that indicates the start of the main code section.

• **Check statements and orders**: Checks the statements and statements that follow the function definitionmain.

• **Return result**: ReturnsTrueif the function definitionmain is valid and valid statements and commands follow, otherwise returns False.

```python
def callMainPart(token):
    global scope
    global pos
    global framelen
    recordStructure.addNewScope()
    if token[0] == "defitoken":
        token = lex()
        if token[0] == "commandtoken":
            if token[2] == "main":
                q = token[2]
                pos=pos+4
                newPos=pos
                newScope=scope-1
                framelen=newScope
                recordStructure.addNewEntity(scopeIndex=newScope,entityName=q,entity_type='Function', framelength=newPos,arguments=[])
                quadList.append(genQuad("begin_block","main","_","_"))
                token=lex()
                if declarations(token):
                    token=lex()
                if statements(token):
                        return True
                else:
                    print ("At line", token[1], "a statement was expected")
                    print("Compilation failed")
                    exit()
            else:
                exit()
        else:
            exit()
    else:
        return False
```

Figure 38

5) Intermediate code

To implement the intermediate code, the auxiliary functions (their implementation is shown in Figure 6) described in the slides and the handbook of the course were used and placed in the appropriate places with the appropriate parameters (according to the handbook), in order to produce the intermediate code . .The description of the above can be seen in chapter 4 in detail.

```python
def nextQuad():
    global quadNum
    return quadNum+1

def genQuad(op,x,y,z):
    global quadNum
    quadNum += 1
    quad = [op,x,y,z]
    return quad

def newTemp():
    global tempNum
    tempNum += 1
    s = "T_"
    return s+str(tempNum)

def emptyList():
    return genQuad("_","_","_","_")

def makeList(label):
    return [label]

def mergeList(list1,list2):
    return list1+list2

def backpatch(lst, label):
    for x in lst:
        quadList[x-1][-1] = label
```

Figure 39

6)Table of symbols

For the symbol table the RecordStructure class was created (Figures 40,41&42) which implements the structure (according to the slides and the handbook) for the symbol table. By using the methods of this structure in the appropriate places the table is created. More specifically, the global variable scope is used which defines the scope of the table and pos which defines the offset. pos is incremented by 4 when an entity is created. The scope increases by 1 when a function is created and decreases by 1 when it reaches its end. The description of the above can be seen in chapter 4 in detail.

```python
class RecordStructure:
    def __init__(self):
        self.scopes = []

    class RecordScope:
        def __init__(self, entityList, nestingLevel):
            self.entityList = entityList
            self.nestingLevel = nestingLevel

    class RecordEntity:
        def __init__(self, name):
            self.name = name

    class Variable(RecordEntity):
        def __init__(self, name,  offset):
            super().__init__(name)
            self.offset = offset

    class Function(RecordEntity):
        def __init__(self, name,  framelength,arguments):
            super().__init__(name)
            self.framelength = framelength
            self.arguments = arguments

    class TemporaryVariable(RecordEntity):
        def __init__(self, name, offset):
            super().__init__(name)
            self.offset = offset

    class RecordArgument:
        def __init__(self, parMode):
            self.parMode = parMode
```

Figure 40

Internal Classes

1.**RecordScope**:
- Represents a scope with a list of entities (entityList) and a level of nesting (nestingLevel).
- __init__(self, entityList, nestingLevel): Manufacturer which initializes the entity list and nesting level.

2.**RecordEntity**:
- The base class for all entities, contains only the name of the entity.
- __init__(self, name): Constructor that initializes the name.

3.**Variable**(inherits fromRecordEntity):

- Represents a variable with a name and an offset (offset).
- __init__(self, name, offset): Constructor that initializes name and offset.

4.**Function**(inherits fromRecordEntity):

- Represents a function with a name, frame length (framelength) and argument list (arguments).
- __init__(self, name, framelength, arguments): Constructor that initializes name, frame length, and arguments.

5.**TemporaryVariable**(inherits fromRecordEntity):

- Represents a temporary variable with a name and an offset (offset).
- __init__(self, name, offset): Constructor that initializes name and offset.

6.**RecordArgument**:

- Represents a function argument with a parameter function (parMode).
- __init__(self, parMode): Constructor that initializes the parameter function.

```python
    def addNewScope(self):
        newScope = RecordStructure.RecordScope(entityList=[], nestingLevel=len(self.scopes) + 1)
        self.scopes.append(newScope)
        return newScope

    def scopeDeletion(self, scopeIndex):
        if scopeIndex < len(self.scopes):
            self.scopes.pop(scopeIndex)


    def addNewEntity(self, scopeIndex, entityName, entity_type, **kwargs):
        if scopeIndex < len(self.scopes):
            if entity_type == 'Variable':
                newEntity = RecordStructure.Variable(name=entityName, **kwargs)
            elif entity_type == 'Function':
                newEntity = RecordStructure.Function(name=entityName, **kwargs)
            elif entity_type == 'TemporaryVariable':
                newEntity = RecordStructure.TemporaryVariable(name=entityName, **kwargs)
            else:
                return None
            self.scopes[scopeIndex].entityList.append(newEntity)
            return newEntity

    def addNewArgument(self, scopeIndex, entityIndex, argumentName, parMode, type):
        if scopeIndex < len(self.scopes) and entityIndex < len(self.scopes[scopeIndex].entityList):
            newArgument = RecordStructure.RecordArgument(parMode=parMode, type=type)
            if not hasattr(self.scopes[scopeIndex].entityList[entityIndex], "arguments"):
                self.scopes[scopeIndex].entityList[entityIndex].arguments = []
            self.scopes[scopeIndex].entityList[entityIndex].arguments.append(newArgument)
            return newArgument

    def searchEntity(self, entityName):
        for scope in self.scopes:
            for entity in scope.entityList:
                if entity.name == entityName:
                    return entity, None
        return None, "Entity not found."
```

Figure 41

Her methodsRecordStructure

1.**__init__(self)**:

- Initializes an empty list of fields (scopes).

2.**addNewScope(self)**:

- Adds a new field to the list of fields.

oReturns the new field.

3.**scopeDeletion(self, scopeIndex)**:

- o Deletes a field from the list of fields according to its index.

4.**addNewEntity(self, scopeIndex, entityName, entity_type, * * kwargs)**:

- o Adds a new entity to a specified field.
- o entity_typemay be 'Variable', 'Function' or 'TemporaryVariable'.
- o Returns the new entity.

5.**addNewArgument(self, scopeIndex, entityIndex, argumentName, parMode, type)**:

- o Adds a new argument to a function within a specified scope and entity.
- o Returns the new argument.

6.**searchEntity(self, entityName)**:

- o Searches for an entity by its name in all fields. Returns the
- o entity and nesting level (orNoneif not found).

7.**printScopesToFile(self,** o **filename="cpy.sym")**:

Prints the fields and entities to a file. Creates a file with a

- o reference to each field and the entities it contains.

```python
    def printScopesToFile(self, filename="cpy.sym"):
        with open(filename, "w") as file:
            file.write("--------------------------------SYMBOL TABLE--------------------------------\n")
            for i, scope in enumerate(self.scopes, start=1):
                file.write(f"Scope {i-1}:\n")
                #file.write(f"Nesting Level: {scope.nestingLevel}\n")
                file.write("Entities:\n")
                for j, entity in enumerate(scope.entityList, start=1):
                    file.write(f"\tEntity {j}:\n")
                    file.write(f"\tName: {entity.name},")
                    if isinstance(entity, self.Variable):
                        file.write(f"\tOffset: {entity.offset}\n")
                    elif isinstance(entity, self.Function):
                        file.write(f"\tFrame Length: {entity.framelength},")
                        file.write(f"\tArguments:{entity.arguments}\n")

                    elif isinstance(entity, self.TemporaryVariable):
                        file.write(f"\tOffset: {entity.offset}\n")
                file.write("\n")
```

Figure 42

7) Final code

For the symbol table, the Final class was created (Figures 43,44,5 & 46) which contains the helper functions of the slides and the handbook (instead of producing there are methods for almost every case). Also created were writeFunctionFinalCode(quadSubList) (generates final code according to the quads it accepts) and readQuadList(quad) (helper to generate the final code) (Figures 47,48,49,50,51 & 52) which help to writing the final code. Because fp does not exist in RISC-V, s0 was used in its place.

```
class Final:
    def __init__(self, record_structure):
        self.record_structure = record_structure
        self.instructions = []

    def gnlvcode(self, var_name, reg):
        entity = self.record_structure.searchEntity(var_name)[0]
        entity_level = self.record_structure.searchEntity(var_name)[1]
        if entity is None:
            print (var_name,"has not been initialized")
            print("Running has failed")
            exit()
        current_level = len(self.record_structure.scopes)
        levels_up = current_level - entity_level
        if levels_up > 0:
            self.instructions.append(f"\tlw {reg}, -4(sp)")
        for _ in range(1, levels_up):
            self.instructions.append(f"\tlw {reg}, -4({reg})")
        self.instructions.append(f"\taddi {reg}, {reg}, {entity.offset}")

    def loadvr(self, v, reg):
        def increment_reg(reg):
            prefix = reg[0]
            num = int(reg[1:])
            return f"{prefix}{num + 1}"

        if str(v).isnumeric():
            self.instructions.append(f"\tli {reg}, {v}")
            return

        entity_level = self.record_structure.searchEntity(v)[1]
        entity = self.record_structure.searchEntity(v)[0]
        if  entity is None:
            print (v,"has not been initialized")
            print("Running has failed")
            exit()

        current_level = len(self.record_structure.scopes)

```

Figure 43

- __init__(self, record_structure): This method is the constructor of the class. Accepts an objectrecord_structure, which probably represents the structure of the source code. Initializes the fieldsrecord_structure, which refers to the structure of the source code, andinstructions, which stores the execution commands.

- gnlvcode(self, var_name, reg): This method generates code that returns the memory address of a variable name. It is mainly used to address the variable in an outer level of scope.

- loadvr(self, v, reg): This method generates code that loads the value of a variable or a constant into a special registerreg. increment_reg(reg) was created to give the next register

- storerv(self, v, reg): This method generates code that stores the value from a special registerregto a variablev.

```
        if entity_level == 1:
            self.instructions.append(f"\tlw {reg}, -{entity.offset}(gp)")
        elif entity_level == current_level:
            if isinstance(entity, RecordStructure.Variable) or isinstance(entity, RecordStructure.TemporaryVariable):
                self.instructions.append(f"\tlw {reg}, -{entity.offset}(sp)")
            elif isinstance(entity, RecordStructure.RecordArgument) and entity.parMode == "CV":
                self.instructions.append(f"\tlw {reg}, -{entity.offset}(sp)")
                self.instructions.append(f"\tlw {reg}, 0({reg})")
                next_reg = increment_reg(reg)
                self.instructions.append(f"\tlw {next_reg}, 0({reg})")
        elif entity_level < current_level:
            self.gnlvcode(v, reg)
            self.instructions.append(f"\tlw {reg}, 0({reg})")
            next_reg = increment_reg(reg)
            self.instructions.append(f"\tlw {next_reg}, 0({reg})")

    def storerv(self, v, reg):
        entity_level = self.record_structure.searchEntity(v)[1]
        entity = self.record_structure.searchEntity(v)[0]
        if  entity is None:
            print (v,"has not been initialized")
            print("Running has failed")
            exit()

        current_level = len(self.record_structure.scopes)
        if entity_level == current_level:
            self.instructions.append(f"\tsw {reg}, -{entity.offset}(sp)")
        elif entity_level < current_level:
            if isinstance(entity_level, RecordStructure.RecordArgument) and entity.parMode == "CV":
                self.gnlvcode(v, reg)
                self.instructions.append(f"\tsw {reg}, 0({reg})")
        elif entity_level == 1:
            self.instructions.append(f"\tsw {reg}, -{entity.offset}(gp)")
```

Figure 44

- end(self): This method generates code that terminates program execution.

- callFun(self, funName): This method generates code that calls a function with the namefunName.

• returnToCaller(self): This method generates code that returns to the calling code from a function.

• jump(self, jumpName): This method generates code that jumps to a tag namedjumpName.

```python
    def end(self):
        self.instructions.append("\tli a0, 0")
        self.instructions.append("\tli a7, 93")
        self.instructions.append("\tecall")

    def callFun(self, funName):
        self.instructions.append(f"\tjal ra, {funName}")

    def retToCaller(self):
        self.instructions.append("\tlw ra,0(sp)")
        self.instructions.append("\tjr ra")

    def jump(self, jumpName):
        self.instructions.append(f"\tj {jumpName}")

    def label(self, labelName):
        self.instructions.append(f"{labelName}:")

    def move(self, r1, r2):
        self.instructions.append(f"\tmv {r1}, {r2}")

    def operations(self, r1, r2, r3, op):
        if op == "+":
            self.instructions.append(f"\tadd {r1}, {r2}, {r3}")
        elif op == "-":
            self.instructions.append(f"\tsub {r1}, {r2}, {r3}")
        elif op == "*":
            self.instructions.append(f"\tmul {r1}, {r2}, {r3}")
        elif op == "/":
            self.instructions.append(f"\tdiv {r1}, {r2}, {r3}")
        elif op == "not":
            self.instructions.append(f"\tnot {r1}, {r2}")
        elif op == "or":
            self.instructions.append(f"\tor {r1}, {r2}")
        elif op == "and":
            self.instructions.append(f"\tand {r1}, {r2}")
```

Figure 45

• label(self, labelName): This method creates a label with the name labelName.

• move(self, r1, r2): This method generates code that copies the value from a registerr2to another registrarr1.

• operations(self, r1, r2, r3, op): This method generates code for various operations (eg addition, subtraction) between the values located in three registers (r1,r2,r3).

```python
    def branch(self, r1, r2, label, con):
        if con == "==":
            self.instructions.append(f"\tbeq {r1}, {r2}, {label}")
        elif con == "!=":
            self.instructions.append(f"\tbne {r1}, {r2}, {label}")
        elif con == ">=":
            self.instructions.append(f"\tblt {r1}, {r2}, {label}")
        elif con == "<":
            self.instructions.append(f"\tbge {r1}, {r2}, {label}")

    def write_instructions(self,s):
        for instr in self.instructions:
            s+=(instr + "\n")
        self.instructions = []
        return s
```

Figure 46

• branch(self, r1, r2, label, con): This method generates code that programmatically implements program execution in case a certain condition is true or false.

• write_instructions(self, s): This method creates a string which contains the execution commands that have been created and are stored in the fieldinstructions

```python
    def writeFunctionFinalCode(quadSubList):
        global finalCode
        global framelen
        branchRelOperators = ["=",">","<","!=",">=","<="]
        arithmeticOperators = ["+","-","*","/","and","not","or"]
        index = 0
        while(index<len(quadSubList)):
            quad = quadSubList[index]
            currentLabel = getNextLabel()
            finalCode += currentLabel+":\n"
            if(quad[0] == "begin_block" and quad[1] == "main"):
                finalCode += "Lmain:\n"


            if(quad[0] == "begin_block"):
                funcBeginLabels[quad[1]] = currentLabel
                finalCode +="\tsw ra,0(sp)\n"

            elif(quad[0] == "end_block"):
                final.retToCaller()
                finalCode = final.write_instructions(finalCode)

            elif(quad[0] == ":="):
                if(recordStructure.searchEntity(quad[1]) is not None):
                    final.loadvr(str(quad[1]),"t0")
                    finalCode = final.write_instructions(finalCode)
                else:
                    final.gnlvcode(str(quad[1]),"t0")
                    finalCode = final.write_instructions(finalCode)

                final.storerv((quad[3]),"t0")
                finalCode = final.write_instructions(finalCode)

            elif(quad[0] in branchRelOperators):
                final.loadvr(str(quad[1]), "t0")
                finalCode = final.write_instructions(finalCode)
                final.loadvr(str(quad[2]), "t1")
                finalCode = final.write_instructions(finalCode)
                final.branch("t0","t1","L"+str(quad[3]),str(quad[0]))
                finalCode = final.write_instructions(finalCode)

```

Figure 47

```
elif(quad[0] == "jump"):
    finalCode +=("\tj L"+str(quad[3])+"\n")

elif(quad[0] == "ret"):
    finalCode += "\tlw t0 -8(sp)\n\tlw t1, -"+str(pos)+"(sp)\n\tsw t1, 0(t0)\n"

elif(quad[0] in arithmeticOperators):
    if(str(quad[1]).isnumeric()):
        if(quad[0] == "+" or quad[0] == "-"):
            finalCode += "\taddi t0, zero, "+str(quad[1])+"\n"
        elif(quad[0] == "*" or quad[0] == "/"):
            finalCode += "\muli t0, "+str(quad[1])+"\n"
    else:
        final.loadvr(str(quad[1]),"t0")
        finalCode = final.write_instructions(finalCode)

    if(str(quad[2]).isnumeric()):
        if(quad[0] == "+" or quad[0] == "-"):
            finalCode += "\taddi t1, zero, "+str(quad[2])+"\n"
        elif(quad[0] == "*" or quad[0] == "/"):
            finalCode += "\muli t1, "+str(quad[2])+"\n"
    else:
        final.loadvr(str(quad[2]),"t1")
        finalCode = final.write_instructions(finalCode)

    final.operations("t0","t1","t2",str(quad[0]))
    finalCode = final.write_instructions(finalCode)
    final.storerv(str(quad[3]),"t2")
    finalCode = final.write_instructions(finalCode)
```

Figure 48

```
elif(quad[0] == "mod"):
    if(recordStructure.searchEntity(quad[1]) is not None):
        final.loadvr(str(quad[1]),"t0")
        finalCode = final.write_instructions(finalCode)
    else:
        final.gnlvcode(str(quad[1]),"t0")
        finalCode = final.write_instructions(finalCode)

    if(recordStructure.searchEntity(quad[2]) is not None):
        final.loadvr(str(quad[2]),"t1")
        finalCode = final.write_instructions(finalCode)
    else:
        final.gnlvcode(str(quad[2]),"t1")
        finalCode = final.write_instructions(finalCode)

    final.loadvr(str(quad[3]),"t2")
    finalCode = final.write_instructions(finalCode)
    finalCode+= "\trem t2,t0,t1\n"
    final.storerv(str(quad[3]),"t2")
    finalCode = final.write_instructions(finalCode)

elif(quad[0] == "inp"):
```

Figure 49

```python
            elif(quad[0] == "inp"):
                final.loadvr(str(quad[1]),"t0")
                finalCode = final.write_instructions(finalCode)
                finalCode+= "\tli a0, 0\n\tli a2, 1\n\tli a7,63\n\tecall\n"
                final.move("t0","a0")
                finalCode = final.write_instructions(finalCode)

            elif(quad[0] == "out"):
                final.loadvr(str(quad[1]),"t0")
                finalCode = final.write_instructions(finalCode)
                finalCode+= "\tlw a0, 0(t0)\n\tli a7,1\n\tecall\n"

            elif(quad[0] == "halt"):
                final.end()
                finalCode = final.write_instructions(finalCode)

            elif(quad[0] == "par"):
                finalCode+= "\taddi s0,sp,"+str(framelen)+"\n"#### opou fp s0
                while(quad[0] =="par" and quad[2]!="ret"):
                    entity,entity_level = recordStructure.searchEntity(quad[1])
                    offset = entity.offset
                    finalCode += getNextLabel()+":\n"
                    final.loadvr(str(quad[1]),"t0")
                    finalCode = final.write_instructions(finalCode)
                    finalCode+= "\tsw t0, -"+str(offset)+"(s0)\n"###opou fp s0
                    index+=1
                    quad = quadSubList[index]


                finalCode += getNextLabel()+":\n"
                finalCode+= "\taddi t0, sp, -"+str(offset)+"\n"
                finalCode+= "\tsw t0, -8(s0)\n"###opou fp s0
                index+=1
                quad = quadSubList[index]
                finalCode += getNextLabel()+":\n"
                label = funcBeginLabels[quad[1]]
                final.callFun(label)
                finalCode = final.write_instructions(finalCode)
            index+=1
```

Figure 52

```python
def readQuadList(quad):
    funcName = quad[1]
    endIndex = quadList.index(quad)
    beginIndex = quadList.index(["begin_block",funcName,"_","_"])
    writeFunctionFinalCode(quadList[beginIndex:endIndex+1])
```

Figure 52